Responsibilities and rewards: Reasoning about design patterns

Neelam Soundarajan and Jason Hallstrom
Computer and Information Science
Ohio State University, Columbus, OH 43210
e-mail: {neelam, hallstro}@cis.ohio-state.edu

Abstract

Design patterns provide guidance to system designers on how to structure individual classes or groups of classes, as well as constraints on the interactions between these classes, to enable them to implement flexible and reliable systems. Patterns are usually described informally. While such informal descriptions are useful and even essential, if we want to be sure that designers precisely and unambiguously understand the requirements that must be met when applying a given pattern, and be able to reliably predict the behaviors the resulting system will exhibit, we also need formal characterizations of the patterns.

In this paper, we develop an approach to formalizing design patterns. The requirements that a designer must meet with respect to the structures of the classes, as well as with respect to the behaviors exhibited by the relevant methods, are captured in the responsibilities *component of the pattern's specification; the benefits that will result by applying the pattern, in terms of specific behaviors that the resulting system will be guaranteed to exhibit, are captured in the* rewards *component. One important aspect of many design patterns is their flexibility; our approach is designed to ensure that this flexibility is retained in the formalization of the pattern. We illustrate the approach by applying it to a standard design pattern.*

1 Introduction

Design patterns [1, 2, 5, 6, 13] are widely regarded as one of the most powerful tools in designing OO systems. Specific patterns provide precise guidance on how to structure individual classes or groups of classes, as well as the interactions between these classes, to implement flexible and easily extensible solutions in specific contexts. Using design patterns helps system designers in two ways: First, design patterns represent the collective wisdom and experience of the community, so the application of the relevant patterns should lead to higher quality systems with predictable behavior. Second, when a new designer joins the design team, he or she will be able, given prior knowledge and understanding of the specific patterns involved, to get a quicker and more thorough grasp of how the system is structured, why it behaves in particular ways, etc.

But if we are to fully realize these benefits, we must have precise ways to *reason* about these design patterns that we can apply to understand and predict the behavior of systems built using them. Our goal in this paper is to investigate such reasoning techniques. In our approach, the specification of a pattern will consist of two components. The *responsibilities* component will consist of the conditions that a designer must ensure are satisfied with regards to the structures of the classes, the behaviors of particular methods of the classes, and the interactions between various classes. The *rewards* component will specify the particular patterns of behaviors that the resulting system will be guaranteed to exhibit if all the requirements contained in the responsibilities component are satisfied.

In specifying a pattern, we will start by listing the *roles* that make up the pattern. For example, in the case of the Observer pattern [5], which we will use as the running example throughout the paper, there are two *roles*, Subject and Observer. This information is already clear from the standard UML representation of the pattern (Fig. 1). Also clear from the diagram is the fact that the Notify() method of Subject will invoke the Update() method on each of the observers[1]. What is not clear is when Notify() will be called and by whom. This question is addressed in the commentary (under "Collaborations") in the description of the Observer pattern in [5], "… subject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own." But it is not clear how the

[1]We use names starting with uppercase letters such as Subject for roles; and the corresponding lowercase names such as subject to refer to the individual objects that play these roles. We also use names starting with uppercase letters for patterns, classes, and methods; member variables will have names starting with lowercase letters. In some cases the name of a pattern is also used for one of the roles in that pattern, as in the case of the Observer role of the Observer pattern. In such cases, the context will make clear whether we are talking about the role or the pattern.

subject will know when its state has become inconsistent with that of one or more of the observers. Indeed, what does it mean to say that the subject state has become *inconsistent* with that of an observer? How is this to be determined?

Perhaps more to the point from the point of view of a designer, what exactly are the requirements that the designer must ensure are met in order to apply the pattern as intended? And what exactly can the designer expect in return for meeting these requirements? For the Observer pattern, the return is that the if the pattern is applied as intended, the view of each observer will remain consistent with the subject's state – again consistent in what sense? As we will see, the kind of formal specifications that we develop in this paper will provide precise answers to these questions.

There is an inherent risk in formalizing design patterns [12] in that *flexibility*, which is the hallmark of many design patterns, may be lost. For example, in the case of Observer, if we adopt one definition for the notion of *inconsistency*, the pattern may not be usable in systems that have a different notion of this concept. As we will see, our approach, while requiring us to provide a precise characterization of the pattern, also enables us to retain the flexibility contained in the pattern.

There is a similarity between our approach to specifying design patterns and the standard design-by-contract (DBC) approach [7, 8, 9] using pre/post-conditions and invariants to specify classes and methods. In DBC, the methods of each class have a precisely defined pre-condition and post-condition, and the class as a whole has an invariant associated with it. Whenever invoking a method of a class, it is the caller's responsibility to make sure that the corresponding pre-condition is satisfied. In return, the caller is assured that when control returns, the post-condition will be satisfied (assuming that the method is implemented correctly). In our formalization of patterns, the specification will specify a set of requirements that the system designer wishing to use this pattern must meet; in return, the designer is assured that the invariant associated with the pattern will be satisfied (as long as the 'pattern instance' –a notion to be defined shortly– continues to exist). In a sense, patterns are a natural extension of classes; while classes focus on how individual objects should behave, patterns are concerned with interactions between a group of objects and relations between their respective states. Note that it may well be possible to encapsulate this entire group of objects into a class and focus on the interactions of this class with *external* objects. If we did that, we would again be in the domain of standard DBC. But here we want to think of the objects in this group as individual entities and consider the interactions *between* them. DBC allows us to deal with individual classes in a precise manner; our work is an attempt to extend that to deal with the group of objects that a given pattern is concerned with.

Most previous work on patterns has focused on identifying patterns and specifying them informally, usually in the style introduced by [5]. There has been some work related to formal aspects of patterns; these have dealt with such things as modeling patterns using temporal action systems, categories, etc. To our knowledge, there has not been much work that has tried to specify patterns by formally defining the exact program properties that the use of the pattern will ensure, or the requirements that a system adopting a given pattern should meet. We will briefly address individual items of related work later in the paper but one important notion that we should mention here is Reenskaug's [11] *role models*. Role models allow us to focus on the interactions between the various objects participating in what he calls an *area of concern*. For example, an area of concern for a system could be the question of how a group of observer objects keep track of the changes in a subject. An important point here is that a given object playing the role of an Observer may have many other aspects to it that are not relevant to this area of concern. Although Reenskaug's work is not concerned with formal specifications, his ideas on role models have heavily influenced our approach to specifying patterns formally. As we will see, the major part of the pattern's specification is the precise specifications of the various roles in that pattern.

The rest of the paper is organized as follows: In the next section, we develop the basic ideas of our approach. We analyze the different components that must be included in the specification of a pattern and some possible variations. In Section 3, we apply our approach to the Observer pattern and develop a specification for this pattern. In Section 4 we consider related work. In the final section, we summarize our approach, reiterate the advantages of having formal specifications of patterns, and consider pointers for future work.

2 An Approach to Specifying Patterns

Consider the Observer pattern [5] in Fig. 1. This pattern is intended to be used when one or more ("observer") objects are interested in the state of a ("subject") object. When an object wishes to become an observer of a subject, it invokes the Attach() method of that object; and when it is no longer interested in that subject, it invokes its Detach() method. (One might also ask if a *third party* object can enroll a given object to play the role of a subject or observer; the formal specification in the next section will answer this question.)

When a subject's state changes, it executes the Notify() operation; this results in the Update() operation of each observer being invoked. What the Update() operation does depends on how it is defined in the ConcreteObserver class(es). The goal of the Update()

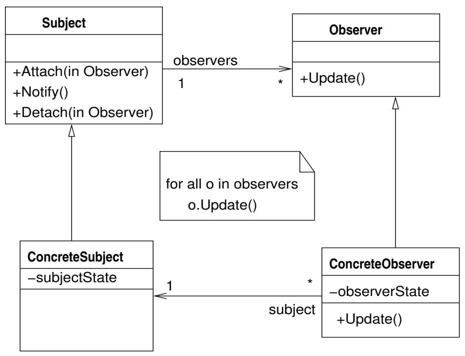

Figure 1. Observer Pattern

operation should be to update the observer's view of the state of the subject so that it is consistent with the actual current state of the subject.

How do we formalize the Observer pattern? Note first that the pattern does not really tell us anything much about the ConcreteSubject or the ConcreteObserver[2]. The focus is on the Subject and the Observer, and the interactions between them. Correspondingly, in our specification of this pattern, we will say that it has two *roles*: subject and observer. It is worth stressing that a role is neither a class nor an object. Instead, objects that wish to interact according to this pattern will *enroll* in the pattern to play the different roles. They will then have to obey the constraints imposed on the respective roles by the pattern's specification.

Before we turn to those constraints, two other points should be noted. First, in a given system, at a given time during execution, it is possible that we have two observer objects O1 and O2 'observing' the subject object S1 while two other objects O3 and O4 are observing a different object S2. In this situation, S1, O1, O2 form one group of interacting objects interacting (if the system has been built correctly) according to the Observer pattern, while S2, O3, O4 is a second group, independent of the first, also interacting according to the same pattern. What we have at this point are two *instances* of the Observer pattern. Thus at any given time, we may have many different instances of any given pattern. Moreover, a pattern instance is not a permanent notion. If, for example, S2 were to cease to exist, then of course O3 and O4 cannot observe it any longer, and the corresponding pattern instance will cease to exist. But, on the other hand, even if both O3 and O4 were destroyed, this pattern instance might continue to exist, with S2 waiting for other observer objects to become interested in it in the future.

[2]In fact, in the original diagram in [5] for Observer [5], some additional information is included about the ConcreteSubject and ConcreteObserver. This information says that ConcreteSubject has a method GetState() and that the ConcreteObserver's Update() invokes that method to get the current state of the subject. But this seems too restrictive. The diagram in [5] further says that *every* observer will set its state (observerState) to the value returned by the GetState() method of subject. Does that mean all observers have the same kind of state? That is clearly not the intention since the whole point of having different observers is to have different views of the state of the subject; but a designer studying that diagram could well get such an impression although the accompanying discussion in [5] should clarify this. In any case, we believe having formal specifications of the kind we propose in this paper for patterns can help in eliminating or reducing such misunderstandings.

Second, apart from the differences in the details of what the two roles in the Observer pattern are responsible for, there is one other important distinction between them which is already implicit in our discussion so far. This is that while any number of objects may play the Observer role in a given instance of the Observer pattern, we must have exactly one object playing the Subject role. Indeed, in order to create a new instance of this pattern, what we need is an object deciding to play the Subject role. Even if no other objects have expressed an interest in observing this subject object, we will create a new instance of the pattern. This instance will wait for objects to enroll in the role observer, but no new object can enroll in the Subject role of this instance.

Each role will be specified by a *requires* clause and an *ensures* clause for each method. In addition, we will also use a *preserves* clause which will just say that the state components specified in that clause are not changed by that method. However, we are dealing with roles here, not classes. This means that we do not know what the complete state of the actual class (of the object that will enroll to play this particular role) will consist of. All we can be sure of is that there must be certain components that will be used to maintain the information dictated by the pattern, and to achieve the interactions required by the pattern between its various roles. We will handle this problem as follows. The pattern specification will include a portion labeled *state*. This will list all the components needed from the point of view of the pattern. These components will be partitioned into states corresponding to the individual roles. The *requires*, *preserves*, and *ensures* clauses will be written in terms of these components. In addition, it may also be appropriate to have a component corresponding to the pattern as whole, i.e., that cannot be logically thought of as being part of one or the other of the roles, but that is not the case with the Observer pattern. Interestingly, when we asked ourselves this question, it occurred to us that there might indeed be a more general version of this pattern that does have such a component. We will briefly mention this generalization in the next section.

But the *state* portion of the pattern specification is not by itself adequate to allow us to specify the various methods of the various roles. Consider, for example, the requirement that Notify() should be invoked when the state of the subject object changes since the views that the observers have of the subject may no longer be consistent with its new state. But what does it mean to say that the state of the subject has changed? It surely should not mean that every time any bit in that object changes, we need to update all the observers. We need to be able to allow the answer to this question to be provided when the object in question enrolls to play the Subject role rather than assuming one particular answer when defining the pattern. To allow for this, in our specification, we use the notion of *auxiliary concepts*. Essentially, we parameterize our specification in terms of these concepts. In the case of the Observer pattern, we will use two such concepts. One will correspond to the notion of how to determine if the state of the subject has changed. The second will answer the question, corresponding to the given subject and each observer object, what does it mean for the observer's state to be consistent with that of the subject. At the point that the pattern instance is created or at the point that an object enrolls to play a particular role in a currently existing pattern instance, the system designer will have to provide suitable definitions for these concepts. For that pattern instance, the specification will use these particular definitions.

The notion of consistency should in fact be on a per-observer basis since the whole point of having multiple observers is that each may adopt a different view of the state of the subject, hence what it means for a given view to be consistent with the state of the subject will clearly depend on the view in question. That the notion of consistency to be used varies from one observer to the next will become evident in our specification where, as part of an object's enrollment in the Observer role, we will have to provide the definition of the corresponding notion of consistency. This also raises the possibility that perhaps the notion of what it means for the subject's state to be modified should also depend in some way on the state of the observer. For example, if a given observer is not interested in a particular part of the subject's state then, from the point of that observer, changes in that portion of the state should not be counted as "modifying the state of the subject"; but a different observer that is interested in this part of the subject's state will want to count such a change as modifying the subject's state. We will return to this point when we develop the formal specification in the next section.

Let us now consider an important issue related to the *ensures* clauses of the individual methods in the different roles. While many of these will be expressible in terms of the corresponding state components, for some we need a more elaborate mechanism. This has to do with the fact that in some situations, the pattern dictates not that a particular method change the state to meet some specified condition, but that during its execution it make a specific sequence of calls to other methods (either other methods of the same role, or methods of other roles). A good example of this is the Notify() method of the subject role of Observer; this method is required, during its execution, to invoke the Update() methods of each observer. To specify such requirements, we will use the *call sequence* or *trace* that records the calls a method makes during its execution and allow conditions to be imposed on this trace, as part of the ensures clause of the method.

So far we have primarily focused on the requirements that objects enrolling to play specific roles must meet, in other words on the *responsibilities* component of the pattern. There are some additional aspects to this which we will see in the next section but for now let us turn to the *rewards* component. In many patterns, the main reward of consistently abiding by the guidelines of the pattern is that an invariant relation between the states of the various objects playing roles in that pattern instance will be satisfied. Thus, for example, in the case of the Observer, this invariant is that whenever control is not inside one of the methods of these objects, the state of each observer will be consistent with that of the subject where consistency is as defined in the corresponding auxiliary concept when this observer object enrolled to play its role. For other patterns, there are other rewards that the designer can expect to reap by applying the pattern correctly. One of the rewards to be expected when using the Memento pattern, for example, is an information hiding guarantee. In future work, we aim to characterize other rewards and to investigate the techniques to specify them.

We conclude this section with a comment about the relation between the *code* of the system and the pattern(s) it uses. According to our discussion, when an object enrolls, for example, in the Subject role of a new instance of Observer, the designer must provide definitions for the auxiliary concepts, check that the methods of the class that this object is an instance of meet the requirements of the corresponding role specifications, etc. But there is nothing in the code of the system where this object declares its intention of playing the role of Subject in an instance of Observer. Unlike, for example, a class and its methods, the patterns are not explicit in the code; they are, so to speak, in the eyes of the design team. To handle this, we will have to introduce *pattern-code* (which will essentially be written to fit the format of comments of the underlying programming language so that normal compilers can ignore it) at suitable points, in particular at points where a new pattern instance is created as well as at points where an object enrolls in or dis-enrolls from an existing pattern instance. Verification responsibilities will materialize, so to speak, at the points wherever this meta-code exists, requiring a conscientious designer to verify that the appropriate responsibilities had been met at these points. This verification may be fully formal, or semi-formal, or informal depending on the tastes of the designer. Alternately, or in addition, automated tools can be built that can mechanize part of this verification task, or possibly test at run-time whether the appropriate assertions are satisfied at the appropriate points (in the meta-code).

3 Case Study

In this section, we develop the formal specification of the Observer pattern. For convenience, we have split the specification into three separate figures. The first one, Fig. 2, is concerned with the pattern-level portion of the specification. The second one, Fig. 3, is concerned with the specification of the Subject role; and Fig. 3 with the specification of the Observer role.

```
pattern Observer {
  roles: Subject, Observer*;            // see note 1 below
  state:
    Subject: set[Observer] _observers;          // note 2
    Observer: Subject _subject;
    pattern: null;
  auxiliaryConcepts:                            // note 3
    relation: Consistent(Subject.$\alpha\sigma$, Observer.$\alpha\sigma$);
    relation: Modified(Subject.$\alpha\sigma_1$, Subject.$\alpha\sigma_2$);
    constraint:
      [$\neg$Modified(su$\alpha\sigma_1$, su$\alpha\sigma_2$)
          $\wedge$ Consistent(su$\alpha\sigma_1$, ob$\alpha\sigma$)] $\Rightarrow$
                Consistent(su$\alpha\sigma_2$, ob$\alpha\sigma$)
  invariant:                                    // note 4
    $\forall$ ob $\in$ _observers: Consistent(ob.$\alpha\sigma$, subject.$\alpha\sigma$)
  pattern instantiation:                         //note 5
    $\langle$Subject:player, Modified$\rangle$
  enrolling as Subject: $\langle$false$\rangle$
  enrolling as Observer:
    $\langle$[(player $\neq$ Subject.player)
          $\wedge$ (player $\notin$ _observers)],
     Consistent, Subject.player.Attach(self)$\rangle$
```

Figure 2. Observer Specification: Pattern

Notes:

1. There are two roles, Subject and Observer. The '*' at the end of the Observer says that any number of objects may enroll to play this role in a given instance of this pattern; by contrast, only one object may play the Subject role.

2. The state is partitioned into a component corresponding to each role. The part corresponding to the Subject role consists of a variable, _observers, which is a *set* of references to observer objects, these of course being the objects that have enrolled to play that role. The part corresponding to the Observer role is _subject which will hold a reference to the object playing that role; note that since multiple objects may enroll to play the Observer role, this state component will be duplicated, one corresponding to each such observer object. In this pattern, there is no component corresponding to the pattern as a whole, hence this part is specified to be null. One possible generalization would be to have a pattern in which there is a limit on the number of observers in any given instance of the pattern. In that case, we could keep a count of the number of objects currently enrolled to play this role, and one of the conditions for enrolling as an Observer would be that the value of count be less than the limit.

   These are not the 'complete' states of the actual subject or observer objects. Those objects will have additional components (defined in their respective classes), and we will have to refer them in our pattern specification. For example, we will have to state that when the state of the subject changes, the observers will be notified. But what do we mean by the state of the subject? This is not the part of the state that is listed here as belonging to the subject role. This part has to do with the state needed by the *pattern*. The actual object that plays this role will have additional state. Indeed, if it didn't, there would be nothing for the observers to observe! We will use the notation $\alpha\sigma$ (for "additional state") to refer to this non-pattern portion of the object's state.

3. The notion of what it means for the state of an observer to be Consistent with a state of the subject depends on the pattern instance and the subject and observer in question, hence it is listed as an auxiliaryConcept. Similarly, given two subject states (which, at the point where we use it in Fig. 3, will be the state at the start and end of a method in subject's class), the Modified concept lets us decide whether the second state is modified or unmodified from the first. These concepts will be defined when an object enrolls as an observer and an object enrolls as a subject at pattern instantiation time respectively (note (5) below).

   But there is an additional condition we need to impose. Suppose we have two subject states, $su\alpha\sigma_1, su\alpha\sigma_2$ that, according to the definition of Modified are the 'same'; suppose also that there is an observer state $ob\sigma$ that, according to the definition of Consistent is consistent with $su\alpha\sigma_1$ but *not* with $su\alpha\sigma_2$. Then we will have a problem because if the subject state were to change from $su\alpha\sigma_1$ to $su\alpha\sigma_2$, the subject would not call the Update() methods of the observers since its state has 'not changed' according to the given Modified; but if the state of one of the observers is $ob\alpha\sigma$, this would no longer be consistent with the new state of the subject! The specified constraint ensures that this problem will not arise. It may be worth pointing out that since the Consistent concept depends on the particular observer ob in question, it may be more accurate to use the notation $\text{Consistent}_{\text{ob}}$; we omit the subscript in the interest of brevity.

4. The pattern invariant is the reward for using this pattern. This asserts that for each observer in the set _observers, its state will be consistent with that of the state of the object playing the Subject role.

5. Next we specify what is needed for creating a pattern instance and for enrolling in the Subject and Observer roles. To create a new pattern instance, we are required, according to this specification, to provide an object that will play the Subject role, which means that the class of that object must meet the constraints (in Fig. 3) for that role; and to provide the definition for Modified, one of the two auxiliary concepts. (Note that player is a keyword that represents the object that wishes to play the role of Subject in the pattern instance about to be created.)

   For enrolling as a Subject after the pattern instance has been created, we are required to satisfy the impossible condition false. This simply means that another object cannot decide to enroll into this role once the pattern instance has been created.

   The conditions for enrolling as an observer are of course more liberal. The requirements are that the object (player) that wants to play this role must not already be enrolled (in this instance) as an observer or subject. Further, when an object enrolls to play this role, we must provide a definition for the auxiliary concept Consistent; this definition is the one we referred to as $\text{Consistent}_{\text{ob}}$ above, and will be used only for this observer object. The final component states that to enroll as an observer, the object needs to invoke the Attach() method on the object playing the Subject role, and pass itself as argument.

   In addition, when creating a pattern instance or enrolling into an existing pattern, we are also required to provide mappings from the (class of the) actual object enrolling to play the particular role to the corresponding *role state*.

Some points are worth noting. The potential incompatibility between the definitions of Modified and Consistent that we discuss in point (3) above and the solution we have proposed, in the form of the corresponding constraint seems to be new. We have not seen this potential problem mentioned before and indeed, it was only as we were designing the formal specification in Fig. 2 that we realized the problem. A less significant point has to do with the conditions to be met when an object wishes to enroll in an existing instance of the Observer pattern. This condition too seems to have been overlooked in much of the literature, the one exception we are aware of being [10] which gives a temporal characterization of this condition. The point is that formal specifications while they are certainly challenging to write down, allow us to identify potential problems that can be overlooked in the informal documentations, and to resolve them. The goal, of course, is to build systems that are more reliable.

```
roleSpec Subject {
  initCondition _observers = $\Phi$;                         // note 6
  invariant true;
  methods:                                                  //note 7
    void Attach(Observer ob):
      requires: $(ob \notin \_observers) \wedge (ob \neq this)$
                $\wedge (ob = caller)$
      preserves: ob; $\alpha\sigma$;                                 //note 8
      ensures: _observers =
                  _observers@pre $\cup$ {ob};                    //note 9
      ensures (call sequence): call ob.Update()
    void Detach(Observer ob):
      requires: ob $\in$ _observers; ob = caller
      preserves: ob; $\alpha\sigma$;
      ensures: _observers = _observers@pre $-$ {ob};
    void Notify():                                          //note 10
      requires: true
      preserves: ob; _observers; $\alpha\sigma$;
      ensures (call sequence):
        calls to ob.Update() for each ob $\in$ _observers
    others:                                                 //note 11
      requires: true
      preserves: _observers
      ensures (call sequence):
        [$\neg$Modified($\alpha\sigma$@pre, $\alpha\sigma$)] $\vee$
        [trace has a call to Notify() $\wedge$
          $\neg$Modified($\alpha\sigma'$, $\alpha\sigma$)] (where $\alpha\sigma'$ is the state
                                          recorded in that call)
}
```

Figure 3. Observer Specification: Subject

Next we turn to the specification of the Subject role in Figure 3.

6. We first specify the initial condition that must be satisfied when an object enrolls into this role (possibly

at the time of the instance creation). Here the condition states that the _observers component of the object must be empty (since no objects should have been enrolled to play the role of observer at this point). There is no invariant for this role, so this is defined to be true.

7. Next we have specifications for the individual methods of this role. Attach(), as we saw above, is the method an object wishing to become an observer should call. Its pre-condition requires that this object not already be an observer; that this object not be the subject; and finally that the object calling Attach() be the same as the one that wishes to become an observer. In other words, this clause prevents enrollment of a given object as an observer of our subject by a third object. If we want to allow for such third-party enrollments, this clause should be omitted.

   We should note that in this pattern, enrollments into roles is a simple matter. But in more complex patterns it may be that enrollment of an object into a particular role takes place only when a number of specified actions, perhaps by a number of specified objects (not just the object enrolling), are completed. Specifying such enrollment requirements will of course be more complex than that in Fig. 2.

8. The preserves clause tells us that the argument ob will be unchanged. And further that $\alpha\sigma$, the "additional state" of the subject object will also be unchanged. In general, all methods explicitly listed in the specification of any role of the pattern are likely to have a similar clause since these methods are only concerned with performing the activities required by the pattern, hence there is no reason for them to modify any component of the additional state.

9. The first part of the ensures clause simply requires the new observer object to be added to _observers. The second part is a condition on the *call sequence* of the Attach() method. This part requires that the call sequence should consist of a call to ob.Update(). The reason for this is to ensure that once an object becomes an observer, its view is consistent with the state of the subject, as indeed required by the pattern invariant. This was another requirement whose need we discovered only as we were designing this specification. The need for such a call does not seem to have been noted elsewhere in the literature.

10. The Notify() method has a more complex call sequence requirement. Essentially, its ensures clause requires that the Update() method of each of the observers is called.

11. The actual class of the object playing this role may not only have additional state, but also additional methods. The "others" part of the specification is a condition that must be must be met by all such methods (i.e., methods other than Attach(), Detach(), and Notify()). The preserves clause requires that these other methods not make any changes in the value of _observers (else the protocol implemented by the Attach() and Detach() will be clearly compromised). Further, the ensures clause requires that if the additional state is modified (as per the definition of the concept Modified which was provided at the time of the pattern instantiation), then there must be a call to Notify() and no further modifications may be made to the state after that call; in other words, the final state must be same as the state that existed at the time of the call to Notify(). This will ensure that the observer's views are updated if the subject state changes.

It is perhaps worth noting that [5] does point out the importance of ensuring this last requirement. We feel, however, that even if we had been unaware of it at the start, we would have discovered the need for it when formalizing this part of the specification. Let us now turn to the specification of the Observer role in Figure 4.

```
roleSpec Observer {
  initCondition _subject = subject.player;    // note 12
  invariant true;
  methods:
    void Update():                            // note 13
      requires: true;
      preserves: _subject;
      ensures: Consistent(_subject.ασ, this.ασ)
    others:                                   //note 14
      requires: true;
      preserves: _subject;
      ensures: Consistent(_subject.ασ, this.ασ)
}
```

Figure 4. Observer Specification: Observer

12. The initialization condition requires that the _subject field be set appropriately to refer to the object that is playing the role of subject.

13. The specification of Update() says that it should not modify the value of _subject, and that it should update the state of the current object to be consistent (as per the definition of the Consistent relation provided when this object enrolled to play this role) with the state of the object referred to by _subject.

14. For the other methods that might be defined in the ConcreteObserver, the requirement is that they not change the value of _subject and that they leave the state of the current object consistent with the state of the subject.

With respect to this last point, it is worth noting that this permits the state of the observer to be modified so long as the new state is also consistent with the current state of the subject. For example, if observer object in question was one that could be displayed on the screen, we might *iconify* it, so clearly its state would change; but this would be permitted by this specification so long as the definition of Consistent that has been provided is such that the iconified state is consistent with the state of the subject. While, as we noted above, the need to call Notify() of the subject when its state is modified has been explicitly noted in the literature, we have not see any discussion of what changes we can make in the observer object independently of the subject. The auxiliary concept Consistent allows us to give a precise characterization of what changes are permitted.

In terms of sheer size, our formal specification is shorter than typical informal descriptions of the pattern although, of course, eve with some practice, it is harder to read than informal descriptions. More important is the precision our specification ensures; it was this precision that allowed us to pinpoint various omissions and gaps in the informal descriptions and allowed us, indeed forced us, to consider ways to fill them.

We have only developed the formal specification of the pattern. In a system that uses one or more instances of the pattern, how do we show that the requirements of the pattern have indeed been met, and how do we make use of the reward, i.e., the pattern invariant specified in Fig. 2? The approach we plan to adopt for addressing these questions is as follows: The designer will be required to include in the body of the system in question, *pattern code* that will represent such actions as a new pattern instance being created, an object enrolling in a given role, etc. Thus, for example, at the point that we decide to create an instance of Observer with a given object playing the Subject role, we would include a line of code such as,

```
/*@ Observer pI = new Observer
        ( Subject: xx,   Modified: (xx.u′ < xx.u) ) */
```

where xx refers to the object that is being enrolled to play the role of Subject in the new instance of Observer being created; the Modified predicate is defined to be the relation specified, where u is a member variable of the class of xx. Recall that Modified is to be a relation between two states of a given object, here the one that xx refers to; when a method operates on this object, it may change the values of one or more member variables of that object; the "′ refers to the value of the variable in question when the operation started and the unprimed variable its value when the operation finishes. Thus here we are saying that, for the purposes of this instance of Observer, we will treat the state of the object to have been modified if the value of xx.u has increased; if the value of xx.u has decreased or not changed, the object will be considered unmodified, independent of what might have happened to the values of the other member variables. In practice, we would probably have more involved notions of what it means for the state to be modified. More important, in practice, we would not want to refer to the states of objects in terms of the values of its concrete member variables, but rather in terms of its *abstract* state. pI is a "pattern variable" that will allow us to refer to this pattern instance later in the program. We need to be able to do this, for example, when another object wishes to enroll as an observer in this particular pattern instance.

Each piece of pattern code will have an association proof obligation. Thus, for the line of code corresponding to creating the new instance pI, we would have to show that the initialization requirements of Fig. 2 are satisfied. In each line that corresponds to an object enrolling as an observer, we would have to show that the definition of Consistent given at that point, and of Modified given above, together satisfy the constraint specified in the pattern specification in Fig. 2. It is by meeting these proof obligations (formally or informally depending on our tastes and the needs of the system), that we would show that we have applied the pattern *correctly*. And, in return, we can assert that as long as pI exists, the invariant specified in Fig. 2 will be satisfied (whenever control is outside all the methods that modify any of the objects enrolled in that pattern instance).

4 Related Work

As we noted earlier, most of the work having to do with design patterns is concerned with informally documenting specific patterns. Dong [3] considers the question of how the use of a design pattern in a particular application can be made clear in a UML diagram. The idea is to annotate the UML diagram so that for any given class, we can specify, if it is part of a design pattern instance. The name of the pattern as well as the "role" (in the pattern) that this class plays are shown in the diagram. The individual operations of the class can similarly carry annotations specifying the roles they play in the application of the pattern. Dong accounts for the fact that the same class might participate in the use of several patterns. But Dong's work does not directly address the question of specifying precisely the pattern in question or the responsibilities that a designer using the pattern must meet.

Eden *et al.* [4] propose a higher order logic formalism in which a design pattern is a formula. The primitives of the logic are things like classes, methods, etc. The key point is that the logic is rich enough to specify the relations that are common *between* patterns. It is not clear whether this formalism will be of use in specifying an individual patterns and identifying, to the designer, the specific responsibilities and rewards that the use of the patterns entails.

Mikkonen [10] specifies a pattern by specifying the classes involved, including an abstract model and a listing of the relations among the various items. Thus for the Observer pattern, he specifies that Attached is a relation between a subject and an observer. He uses an action-system type of specification using a guarded-command language, to specify the sequences of actions formally. Some of Mikkonen's notations, can we believe, be used to simplify the specification that involve the *call sequence* and we plan to investigate this in the future. But for the specifications that do not require the call sequence, using action-system-based specifications makes them hard to understand and reason about.

We have already mentioned the *role models* of Reenskaug[11]. Riehle [12] extends role models to what he calls *pattern composition*. Essentially the idea is that larger patterns can be composed from smaller patterns, where the larger pattern gains its importance as a result of *synergy* between its components, and indeed without such synergy Riehle does not consider the result a composite pattern. His focus is on trying to identify the key roles in the composite pattern and relate them to the roles in the component patterns. While the identification of the roles is done semi-formally, Riehle does not consider the possibility of specifying the expected behaviors of these roles formally which, of course, has been our focus.

5 Discussion

The goal of our work was to develop a technique for reasoning precisely about design patterns and their application. First, we extended the notion of design-by-contract to precisely specify a pattern's behavior. Second, we provided a brief overview of reasoning rules that can be applied to verify the correct application of a pattern, and therefore guarantee the behavioral properties described by the pattern's specification. In this paper we demonstrated our approach in the context of the Observer pattern. The pattern serves as a natural starting point for exploring reasoning issues, as it manifests a number of interesting behavioral properties. Moreover, the pattern is widely used in commercial class libraries like Java Swing and Microsoft's .Net framework.

Recall that one of the benefits of using design patterns is to facilitate in the understanding of complex architectures. When a designer is presented with a new system design, knowledge of the design patterns that went into its construction should enable the designer to understand the structure of the system more quickly and more thoroughly. Without a precise specification, however, the behavior of each pattern participant, and consequently the pattern as a whole, is often ambiguous. In trying to understand the behavior of the Observer pattern so that it could be formally specified, we were confronted with this ambiguity in a number of contexts.

First, when writing the specification of Attach, we realized that when an object enrolls as an observer, it must update its state to be consistent with the current state of the subject. We can't say, for example, wait until the subject state changes, at which point Notify will be called, bringing the observer into a consistent state. What if the subject's state never changes? The observer's view will forever be inconsistent with the subject's state, which of course violates the whole point of using the pattern. It was only when we were writing the formal specification of the pattern that this became clear.

Second, in writing the specification of Detach, there was a question of whether the observer's subject field should be set to null, so that it could no longer interact with the subject. The pattern does not strictly require this behavior, since it is only concerned with observers that are attached to the subject. However, two designers reading the same design documentation might assume different system behaviors when they see that the system uses the Observer pattern. This misunderstanding could lead to a number of problems that could be avoided with the benefit of formal specifications.

We conclude the paper with a couple of pointers to future work. First, we plan to further develop the reasoning rules that can be used to formally verify the correct use of a pattern. These rules will form the basis for a formal proof system for verifying the correct application of a pattern, as well as predicting the behavior that a system will exhibit when developed using the pattern. Another interesting possibility is to implement a monitoring system, *à la* the assertion monitoring system of *Eiffel* [9], that checks, at appropriate points (in the meta-code discussed in Section 2) whether the corresponding responsibility assertions as specified in the pattern are indeed satisfied. While a conscientious designer should check that all responsibilites are indeed met, having such a tool would simplify this task, especially by identifying points where there is a problem. Developing such a tool is also work for the future.

References

[1] K. Beck and R. Johnson. Patterns generate architectures. In *Proceedings of the Eighth ECOOP*, pages 139–149, 1994.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A system of patterns*. Wiley, 1996.

[3] J. Dong. UML extenstions for design pattern compositions. In C. Mingins, editor, *Proc. of TOOLS, in Special Issue of*

Journal of Object Technology, vol. 1, no. 3, pages 149–161, 2002.

[4] A. Eden, J. Gil, Y. Hirshfeld, and A. Yehudai. Toward a mathematical foundation for design patterns. Technical Report 004, Tel Aviv University, 1999.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable OO Software*. Addison-Wesley, 1995.

[6] R. Johnson. Components, frameworks, patterns. In *ACM SIGSOFT Symposium on Software Reusability*, pages 10–17, 1997.

[7] G. Leavens and W. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32:705–778, 1995.

[8] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. on Prog. Lang. and Systems*, 16:1811–1841, 1994.

[9] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[10] T. Mikkonen. Formalizing design patterns. In *Proceedings of 20th ICSE*, pages 115–124. IEEE Computer Society Press, 1998.

[11] T. Reenskaug. *Working with objects*. Prentice-Hall, 1996.

[12] D. Riehle. Composite design patterns. In *Proc. of OOPSLA*, pages 218–228. ACM, 1997.

[13] D. Riehle and H. Zullighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.